

Faire converger du texte

CRDT, identifiants stables et pierres tombales

demi-treillis

RGA

horloge de Lamport

tombstone

local-first

Deux personnes écrivent chacune de leur côté, sans réseau. Comment leurs copies peuvent-elles finir identiques, alors qu'aucune n'a « raison » sur l'autre ?

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Éditeur concurrent CRDT » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Dans le projet « éditeur concurrent », vous complétez trois fonctions pures d'un document répliqué : insérer une lettre, en masquer une, fusionner deux copies. Vous observez ensuite, dans l'interface fournie, qu'Alice et Bob retrouvent le même texte quel que soit l'ordre dans lequel leurs snapshots circulent, même après un doublon ou un message ancien. Le compte rendu vous demande de relier ce résultat aux trois lois de fusion déjà rencontrées en AP8 et à l'hypothèse d'identifiants compatibles.

Ce fascicule ne contient aucune de ces trois fonctions. Il raconte ce qu'il y a autour : d'où vient l'idée d'une structure de données qui se fusionne sans coordination, pourquoi les trois lois suffisent et ce qu'elles cachent, comment une poignée d'algorithmes, de 2006 à 2023, ont donné une identité aux caractères d'un texte, ce que mesurent vraiment les « horloges » du projet, pourquoi les lettres masquées ne peuvent pas être jetées, et ce que deviennent ces idées dans Riak, Phoenix, Figma ou les applications dites *local-first*. Vous pouvez le lire avant, pendant ou après le projet : il n'y a rien à rendre.

2 Converger sans se coordonner

Le compteur d'AP8 et le document d'AP9 partagent une promesse : des copies qui divergent librement, puis se rejoignent dès qu'elles ont échangé. Cette promesse a un nom précis depuis 2011, et une famille de structures de données la tient.

2.1 La cohérence forte à terme

Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski publient en 2011 l'article qui nomme les *CRDT* (*Conflict-free Replicated Data Types*) et définit la garantie qu'ils apportent, la *cohérence forte à terme* (*strong eventual consistency*). Elle réunit trois clauses. La première est la *livraison à terme* : toute mise à jour appliquée sur une réplique finit par atteindre toutes les répliques correctes. La deuxième est la *convergence* : deux répliques qui ont reçu le même ensemble de mises à jour sont dans des états équivalents. La troisième, la *terminaison*, demande que chaque opération se termine.

Le mot « forte » porte sur la deuxième clause. La cohérence à terme ordinaire promet seulement que les répliques finiront par s'accorder, au besoin après une négociation ou un retour en arrière. Ici, l'accord est immédiat dès que les mêmes informations

sont connues : aucun vote, aucun verrou, aucune annulation. C'est exactement ce que vous constatez dans le projet quand les deux textes deviennent $a \vee x b$ à la seconde où le dernier snapshot est livré.

2.2 États ou opérations

L'article de 2011 distingue deux manières de tenir cette promesse. Les CRDT à *états* (en anglais *state-based*, ou CvRDT pour *convergent*) échangent des copies entières de l'état, et chaque réplique fusionne ce qu'elle reçoit avec ce qu'elle a. C'est le choix du compteur d'AP8 et du document d'AP9 : un snapshot est un état complet, et la fusion est la seule porte d'entrée pour une information étrangère. Les CRDT à *opérations* (*operation-based*, ou CmRDT pour *commutative*) diffusent seulement les opérations, « insérer x après a », et exigent du transport qu'il livre chaque opération une fois et dans un ordre compatible avec les causes. En échange, les messages sont petits.

Les deux familles sont équivalentes en pouvoir d'expression : l'article montre comment simuler l'une par l'autre. Elles ne font pas les mêmes hypothèses sur le réseau. Un CRDT à états tolère les doublons, les retards et les pertes rattrapées par un envoi ultérieur, ce que nous avons simulé en AP8 en retenant des snapshots. Un CRDT à opérations délègue ces soucis à une couche de diffusion causale fiable, plus exigeante à construire.

POURQUOI TROIS LOIS SUFFISENT

Supposons que l'état d'une réplique soit l'ensemble des événements qu'elle connaît (entrées insérées, marques de suppression) et que la fusion soit la réunion de deux tels ensembles. La réunion est idempotente, commutative et associative : la réunion d'une collection d'ensembles ne dépend ni de l'ordre, ni des répétitions, ni des regroupements. Deux répliques qui ont appris les mêmes événements, par n'importe quel chemin, ont donc le même état, et la livraison à terme fait le reste. Le point fragile est l'hypothèse : fusionner ne doit jamais perdre un événement connu, et un même identifiant doit toujours désigner le même événement.

3 L'ordre caché derrière la fusion

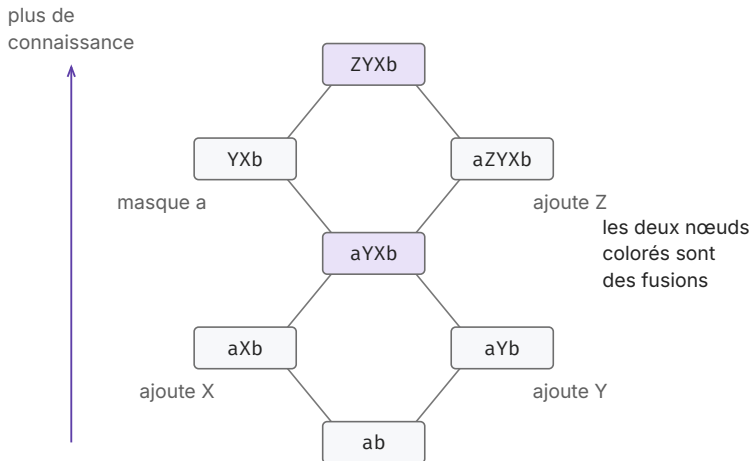
Les trois lois de fusion ne sont pas une liste arbitraire. Elles disent qu'une fusion calcule une *borne supérieure* dans un ordre partiel, et cette lecture explique en une phrase pourquoi un vieux snapshot ne peut rien défaire.

3.1 Un ordre partiel et sa borne supérieure

Posons que l'état a est *plus petit* que l'état b , noté $a \leq b$, lorsque $a \sqcup b = b$: fusionner a dans b n'apprend rien à b . Pour le compteur d'AP8, cela signifie que chaque composante de a est au plus celle de b . Pour le document du projet, cela signifie que les entrées et les marques de a sont incluses dans celles de b . Cette relation est réflexive, antisymétrique et transitive : c'est un *ordre partiel*, « partiel » parce que deux états peuvent être incomparables, par ex. $a \times b$ et $a \forall b$, qui savent chacun quelque chose que l'autre ignore.

Dans cet ordre, $a \sqcup b$ est le plus petit état qui dépasse à la fois a et b : sa *borne supérieure*. Un ensemble ordonné où deux éléments quelconques possèdent une borne supérieure s'appelle un *demi-treillis* (*join-semilattice*). Les trois lois caractérisent exactement cette situation ; réciproquement, dans un demi-treillis, la borne supérieure vérifie les trois lois. Le mot « demi » rappelle qu'on ne demande rien sur les bornes inférieures : nos états savent s'additionner, pas se soustraire.

On dessine un ordre partiel fini par son *diagramme de Hasse* : un nœud par état, un trait de a vers b placé plus haut lorsque $a < b$ et qu'aucun état ne s'intercale. La figure suivante reprend le scénario du projet : le texte commun ab , les deux insertions concurrentes, la suppression d'Alice et l'insertion de Bob.



Le diagramme de Hasse des états du scénario. Monter, c'est en savoir plus. La borne supérieure de deux nœuds est le premier nœud que l'on atteint en montant depuis chacun d'eux ; `aYXb` est celle de `aXb` et `aYb`.

Lisez-le de bas en haut. `ab` est en dessous de `aXb`, qui contient une entrée de plus. `aXb` et `aYb` sont côte à côte, incomparables ; leur borne supérieure est `aYXb`, le seul nœud qui les domine tous deux au plus près. Remarquez le nœud `YXb` : son texte est plus court que celui de `aYXb`, et pourtant il est placé plus haut, car il contient une information de plus, la marque sur `a`. L'ordre porte sur la connaissance, pas sur la longueur affichée. C'est pour cette raison qu'un ancien snapshot, forcément plus bas dans le diagramme, ne fait jamais redescendre une réplique : fusionner avec un état que l'on domine déjà redonne l'état courant.

À RETENIR

Les trois lois de fusion reviennent à dire que la fusion est une borne supérieure dans un ordre partiel. Une mise à jour locale fait monter l'état ; une fusion ne le fait jamais redescendre. Converger, c'est que toutes les répliques finissent au même nœud, le plus haut atteint par l'ensemble des événements.

3.2 Converger n'est pas conserver

Le demi-treillis garantit l'accord, pas la préservation de chaque geste. Le plus simple des CRDT le montre : le registre « dernier écrivain gagne » (*last-writer-wins register*). Son état est un triplet (horloge, écrivain, valeur) ; la fusion garde le plus grand triplet, dans l'ordre des termes d'Elixir. Vérifions les trois lois par énumération sur un petit

domaine, comme les tests du TP le font pour les maps, puis regardons ce que devient la valeur perdante.

```

1 # Un registre « dernier écrivain gagne » :
2 # l'état est {horloge, écrivain, valeur} et la fusion garde
3 # le plus grand triplet, donc une seule valeur.
4 fusion = fn a, b -> max(a, b) end
5 etats =
6   for h <- 1..2, w <- [:alice, :bob], v <- ["x", "y"], do: {h, w, v}
7
8 idempotente = Enum.all?(etats, fn a -> fusion.(a, a) == a end)
9 paires = for a <- etats, b <- etats, do: {a, b}
10 commutative =
11   Enum.all?(paires, fn {a, b} -> fusion.(a, b) == fusion.(b, a) end)
12 triplets = for a <- etats, b <- etats, c <- etats, do: {a, b, c}
13 associative =
14   Enum.all?(triplets, fn {a, b, c} ->
15     fusion.(fusion.(a, b), c) == fusion.(a, fusion.(b, c))
16   end)
17
18 IO.inspect({idempotente, commutative, associative, length(etats)})
19 IO.inspect(fusion.({1, :alice, "x"}, {1, :bob, "y"}))

```

CONSOLE {true, true, true, 8} puis {1, :bob, "y"}

Les huit états forment bien un demi-treillis ; toutes les répliques convergeront. Et la valeur "x" d'Alice a disparu, sans que personne ne l'ait décidé : à horloge égale, l'atome :bob est simplement plus grand que :alice. Un CRDT choisit toujours une politique de résolution ; la convergence dit que tout le monde applique la même, pas qu'elle est juste. Le document du projet a la politique la plus conservatrice qui soit, puisqu'il ne jette rien : toute insertion survit, toute suppression survit, et le prix de ce choix occupe la section 6.

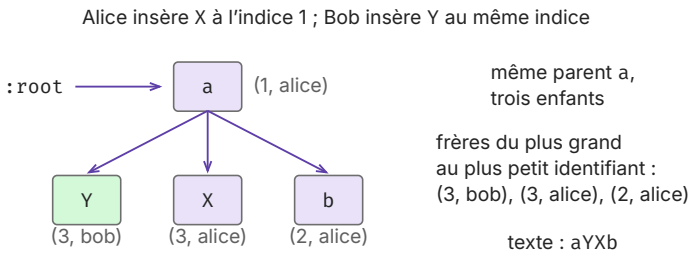
4 Une identité pour chaque caractère

Un compteur se fusionne composante par composante parce que chaque auteur possède sa composante. Un texte n'a rien de tel : « la position 1 » désigne une lettre différente dès que quelqu'un a inséré ou supprimé avant elle. Toute la famille des

CRDT de séquence repose sur une seule décision : remplacer les positions, relatives à une version, par des *identifiants* uniques, absolus, qui ne bougent jamais.

4.1 Le même parent, trois enfants

Dans le projet, chaque entrée porte un identifiant (compteur, écrivain) et l'identifiant de son *ancree*, l'élément visible juste avant elle au moment de l'insertion. Les ancres forment un arbre dont la racine est `:root`. Reprenons les deux insertions concurrentes de la fiche.



Alice et Bob insèrent au même indice sans se voir. Les deux nouvelles entrées désignent le même parent `a` ; le parcours visite un parent puis ses enfants, du plus grand identifiant au plus petit, ce qui donne `aYXb` partout.

Pourquoi « du plus grand au plus petit » et non l'inverse ? Imaginez une seule personne qui tape `ab`, puis revient placer `x` entre les deux lettres. `x` et `b` ont la même ancre `a`, et `x` porte un compteur plus grand puisqu'il est arrivé plus tard. L'ordre décroissant le place en premier : `axb`, ce que l'auteur voulait. Un enfant plus récent se glisse donc toujours tout contre son parent, et les enfants plus anciens sont repoussés vers la droite. Pour deux insertions réellement concurrentes, comme `x` et `y`, le compteur est le même et c'est le nom de l'écrivain qui départage : la règle est arbitraire, mais identique partout, et c'est tout ce qu'il faut pour converger.

Cette construction est celle de *RGA* (*Replicated Growable Array*), publiée par Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim et Joonwon Lee en 2011. Dans RGA, chaque élément porte une estampille unique, une insertion désigne l'estampille de son prédécesseur, les insertions concurrentes sous le même prédécesseur sont rangées par estampille décroissante, et une suppression se contente de marquer l'élément. Le projet en est une adaptation à états : au lieu de diffuser les opérations, nous fusionnons des snapshots, ce qui convient mieux à la coupure simulée d'AP8.

4.2 Quatre façons de nommer une lettre

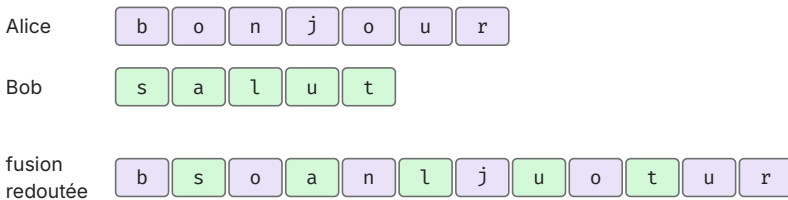
RGA n'est ni le premier ni le dernier de sa famille. *WOOT*, de Gérard Oster, Pascal Urso, Pascal Molli et Abdessamad Imine en 2006, est le premier algorithme

d'édition dont le nom revendique l'absence de transformation (*WithOut Operational Transformation*) : chaque caractère y mémorise les identifiants de ses deux voisins au moment de l'insertion, et l'intégration respecte les deux contraintes. *Treedoc* (Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, Mihai Leția, 2009) et *Logoot* (Stéphane Weiss, Pascal Urso, Pascal Molli, 2009) prennent une autre route : au lieu d'un arbre d'ancres, ils donnent à chaque lettre une *position dense*, un chemin dans un arbre pour Treedoc, une suite de chiffres pour Logoot, telle qu'entre deux positions on peut toujours en fabriquer une troisième. Dans Logoot, supprimer une lettre revient à retirer sa position, sans marque à conserver ; Treedoc doit encore garder la trace d'un nœud supprimé tant qu'il a des descendants. En contrepartie, les positions s'allongent au fil des insertions au même endroit. *LSEQ* (Brice Nédelec, Pascal Molli, Achour Mostefaoui et Emmanuel Desmontils, 2013) est une variante de Logoot qui choisit ses chiffres pour ralentir cet allongement.

Plus récemment, Matthew Weidner et Martin Kleppmann ont proposé *Fugue* (2023), un arbre proche de celui de RGA mais où un nœud peut avoir des enfants à gauche comme à droite. Les auteurs définissent une propriété de « non-entrelacement maximal » et prouvent qu'une variante de leur algorithme la satisfait ; nous y revenons tout de suite. L'article est récent et sa réception se poursuit, aussi le citons-nous comme une piste plutôt que comme un standard.

4.3 L'anomalie d'entrelacement

Converger n'est pas tout. En 2019, Martin Kleppmann, Victor Gomes, Dominic Mulligan et Alastair Beresford décrivent une faute que plusieurs algorithmes convergents commettent : lorsque deux personnes tapent chacune une phrase au même endroit pendant une coupure, certaines fusions mélangent les lettres des deux phrases. L'état final est identique partout, et illisible partout.



les deux mots sont entiers dans l'état, mais mélangés à l'affichage

Alice tape `bonjour` et Bob tape `salut` au même endroit, hors connexion. Une fusion qui ordonne les lettres par identifiant sans tenir compte des ancres peut les alterner. Aucune lettre n'est perdue ; le texte est inutilisable.

Les auteurs montrent que Logoot et LSEQ, qui rangent les lettres par position dense, peuvent produire ce mélange lettre à lettre. RGA s'en sort mieux grâce à ses ancres : quand chaque phrase est tapée dans l'ordre, lettre après lettre, chaque nouvelle lettre a pour ancre la précédente, et les deux phrases restent d'un bloc, dans un ordre ou dans l'autre. Le même article décrit toutefois une « anomalie moindre » propre à RGA. Si Alice tape `reader` après `Hello`, puis revient placer `dear` au même endroit, ces deux insertions partagent une ancre ; une insertion concurrente de Bob, ancrée elle aussi sur `Hello`, peut alors venir se loger entre `dear` et `reader`. Le pire cas, précisent les auteurs, est un texte tapé à l'envers, de la dernière lettre à la première. Fugue est né de cette observation.

UN RÉSULTAT COMMUN N'EST PAS UN RÉSULTAT VOULU

Les tests du projet vérifient que les deux copies affichent le même texte. Aucun test ne peut vérifier que ce texte est celui que les deux personnes auraient écrit ensemble : cette « préservation de l'intention » n'a pas de définition mathématique acceptée par tous. Dans votre compte rendu, distinguez bien ce que les lois garantissent (l'accord) de ce qu'elles ne disent pas (le sens).

5 Des horloges qui comptent sans mesurer

Les identifiants du projet sont des couples (compteur, écrivain), et l'aide fournie choisit toujours un compteur supérieur à tous ceux déjà vus, y compris ceux reçus des autres. Cette règle n'a rien d'une heure : c'est une *horloge logique*, et elle a quarante-huit ans.

5.1 L'horloge de Lamport

Leslie Lamport publie en 1978 un article court dont l'idée est devenue un réflexe de la discipline : dans un système réparti, la seule notion d'ordre dont on dispose vraiment est la relation « a pu influencer ». Un événement *a précède b* (on note $a \rightarrow b$) si *a* arrive avant *b* dans le même processus, si *a* est l'envoi d'un message dont *b* est la réception, ou par transitivité. Deux événements que cette relation ne relie dans aucun sens sont *concurrents* : aucun n'a pu voir l'autre.

Lamport propose alors d'attacher à chaque processus un simple entier qui respecte cette relation : un événement local l'augmente d'un cran ; un message emporte la valeur de son expéditeur ; à la réception, on dépasse à la fois sa propre horloge et

celle du message. Ainsi, si $a \rightarrow b$, alors l'horloge de a est strictement plus petite que celle de b . Voici la règle en trois clauses, rejouée sur Alice et Bob.

```

1  defmodule Lamport do
2    # Un événement local avance l'horloge d'un cran.
3    def local(clock), do: clock + 1
4    # Un envoi est un événement local ;
5    # le message emporte la nouvelle valeur.
6    def send(clock), do: {clock + 1, clock + 1}
7    # Une réception dépasse à la fois sa propre horloge
8    # et celle du message.
9    def receive(clock, stamp), do: max(clock, stamp) + 1
10   end
11
12   alice = Lamport.local(0)
13   {alice, message} = Lamport.send(alice)
14   bob = Lamport.local(0)
15   bob = Lamport.receive(bob, message)
16   IO.inspect({alice, bob, message})

```

CONSOLE {2, 3, 2}

Bob avait tapé une lettre (horloge 1) ; il reçoit un message estampillé 2 et passe à 3. Tout ce qu'il fera ensuite portera un compteur supérieur à tout ce qu'Alice avait fait avant d'envoyer. Reconnaissez-vous la règle du projet ? Un nouvel identifiant dépasse tous les compteurs connus, ceux de la réplique comme ceux des snapshots reçus. C'est une horloge de Lamport, et le nom de l'écrivain accolé au compteur rend chaque couple unique et totalement ordonné. Voilà pourquoi « compteur plus grand » veut bien dire « inséré après avoir vu » : une lettre tapée après la réception de b passera devant b sous la même ancre, comme nous l'avons vu dans la section précédente.

5.2 Les horloges vectorielles et le compteur d'AP8

L'horloge de Lamport a une limite connue : la réciproque est fautive. Deux événements concurrents ont deux entiers, et l'un des deux est plus petit ; l'entier ne permet pas de savoir s'ils se sont vus ou non. Pour *détecter* la concurrence, Colin Fidge et Friedemann Mattern proposent indépendamment, en 1988, de garder un entier par processus : une *horloge vectorielle*. Un vecteur V précède W quand chaque composante de V est au plus celle de W ; deux vecteurs dont aucun ne précède l'autre sont concurrents, et cette fois l'équivalence est exacte.

Vous avez déjà programmé cette structure : le G-Counter d'AP8 est un vecteur indexé par les auteurs, et sa fusion par maximum composante par composante est la borne supérieure de deux vecteurs. Rejouons l'ordre du demi-treillis sur deux snapshots d'AP8.

```
1 defmodule Vecteur do
2   def merge(a, b) do
3     Map.merge(a, b, fn _auteur, x, y -> max(x, y) end)
4   end
5
6   # a est dominé par b quand chaque composante de a
7   # est au plus celle de b.
8   def domine?(a, b) do
9     Enum.all?(a, fn {auteur, n} -> n <= Map.get(b, auteur, 0) end)
10  end
11
12  def concurrents?(a, b) do
13    not domine?(a, b) and not domine?(b, a)
14  end
15 end
16
17 alice = %{alice: 2, bob: 3}
18 bob = %{alice: 1, bob: 4}
19 IO.inspect(Vecteur.domine?(alice, bob))
20 IO.inspect(Vecteur.concurrents?(alice, bob))
21 IO.inspect(Vecteur.merge(alice, bob))
22 IO.inspect(Vecteur.domine?(alice, Vecteur.merge(alice, bob)))
```

```
CONSOLE  false puis true puis %{alice: 2, bob: 4} puis true
```

Alice connaît deux de ses contributions et trois de Bob ; Bob en connaît une d'Alice et quatre des siennes. Aucun snapshot ne domine l'autre : ils sont concurrents, chacun sait quelque chose que l'autre ignore. Leur fusion les domine tous les deux. Les bases de données répliquées utilisent exactement ce test pour distinguer une mise à jour périmée d'une mise à jour concurrente, et c'est sous cette forme, parfois raffinée en *dotted version vectors*, que les horloges vectorielles vivent dans Riak ou dans les anciens systèmes de type Dynamo.

EN PRATIQUE

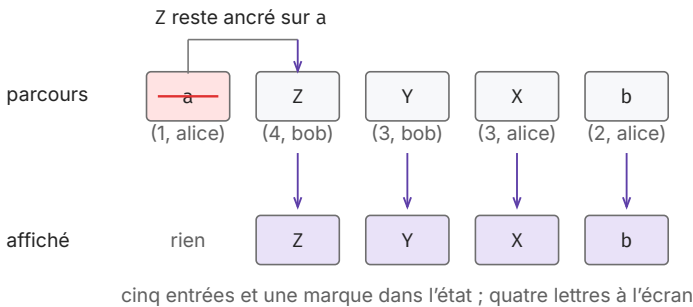
Quand vous construirez vos scénarios du palier 4, notez pour chaque snapshot le plus grand compteur qu'il contient. Un snapshot dont tous les compteurs sont déjà connus de la réplique réceptrice ne peut rien lui apprendre : le prédire avant de cliquer est un bon test de compréhension de l'ordre.

6 Les pierres tombales

Supprimer une lettre dans le projet ajoute son identifiant à l'ensemble `deleted` sans retirer l'entrée. Cette marque s'appelle une *Pierre tombale* (*tombstone*) et elle n'est pas une paresse d'implémentation : c'est la seule manière de garder la fusion correcte.

6.1 Pourquoi on ne peut pas jeter la lettre

Deux raisons l'imposent, et le scénario du projet les montre toutes les deux. D'abord, une insertion encore inconnue peut s'appuyer sur la lettre supprimée : Bob insère `z` avec `a` pour ancre pendant qu'Alice masque `a`. Si Alice avait retiré l'entrée `a`, le snapshot de Bob arriverait avec un `z` orphelin, dont le parent n'existe nulle part. Ensuite, un ancien snapshot peut ramener la lettre : `initial` contient encore `a`. Comme la marque fait partie de l'état et que l'ensemble des marques ne fait que croître, la réunion garde `a` masqué quel que soit le nombre de fois où `initial` est relivré. Retirer l'entrée aurait transformé une suppression en information perdue, et une information perdue ne survit pas à une réunion.



Après la suppression d'Alice et l'insertion de Bob. La ligne du haut est l'ordre du parcours, avec les identifiants ; `a` est parcouru, sert d'ancre à `z`, et n'est pas affiché. La ligne du bas est le texte vu par les deux personnes.

La conséquence est visible dès le palier 4 : masquez tout le texte, réinsérez une lettre, et comparez la longueur affichée à la taille des deux collections. L'état ne rétrécit jamais. Un document longtemps édité porte l'histoire de toutes ses lettres, y compris celles qui n'ont vécu qu'une frappe.

6.2 Le nettoyage demande de savoir ce que les autres savent

Peut-on retirer une pierre tombale un jour ? Oui, à une condition précise : qu'aucune réplique ne puisse plus jamais la mentionner, ni comme ancre d'une insertion encore en route, ni dans un snapshot ancien encore en circulation. Il faut donc savoir que *toutes* les répliques ont vu la suppression et tout ce qui s'y rattache. Carlos Baquero, Paulo Sérgio Almeida et Ali Shoker (2014, version étendue en 2017) appellent cette situation la *stabilité causale* et montrent comment la détecter à partir d'horloges vectorielles échangées entre répliques. Vous voyez le prix : il faut connaître la liste des répliques, et une seule réplique partie sans donner de nouvelles bloque le nettoyage pour toutes les autres. Dans notre projet, « une décision locale ne suffit pas » résume exactement cette contrainte.

Les systèmes réels choisissent entre trois attitudes. Les positions denses de Logoot évitent les marques, au prix d'identifiants qui s'allongent ; les auteurs de Treedoc proposaient déjà en 2009 de rééquilibrer l'arbre, et d'évacuer ses nœuds supprimés, par un accord occasionnel entre répliques. Les bibliothèques qui descendent de RGA, comme Automerge, gardent les marques et travaillent surtout à les encoder de façon compacte. Enfin, beaucoup d'applications acceptent simplement que le document grossisse, parce qu'un texte humain reste petit devant la mémoire d'un téléphone.

7 Au-delà du projet

Les CRDT sont sortis des articles. Nous en citons quelques usages, en nous en tenant à ce que leurs auteurs ont écrit.

La base de données *Riak*, de la société Basho, propose depuis sa version 2.0 (2014) des compteurs, des ensembles, des maps, des registres et des drapeaux que sa documentation décrit comme « fondés sur les types de données répliqués convergents » : le développeur incrémente ou ajoute sur n'importe quel nœud, et la base fusionne. Plus près de vous, *Phoenix*, le framework web d'Elixir, suit la présence des utilisateurs connectés avec `Phoenix.Tracker`, dont la documentation précise que les répliques « utilisent un protocole de battements de cœur et un CRDT pour répliquer l'information de présence à travers la grappe, de manière convergente et

sans conflit ». Quand un site Phoenix affiche « trois personnes regardent cette page » sur plusieurs serveurs, c'est un cousin de votre G-Counter qui travaille.

Pour le texte, deux bibliothèques dominant. *Yjs*, de Kevin Jahns, implémente l'algorithme YATA (Petru Nicolaescu, Kevin Jahns, Michael Derntl et Ralf Klamma, 2016) et équipe de nombreux éditeurs web. *Automerge*, développée autour de Martin Kleppmann et du laboratoire Ink & Switch, descend de RGA et du type JSON répliqué décrit par Kleppmann et Alastair Beresford en 2017. Toutes deux fonctionnent entre pairs, sans serveur obligatoire, et c'est leur argument central face à l'OT.

L'exemple le plus commenté vient de Figma. Dans un billet d'octobre 2019, Evan Wallace, cofondateur, explique que l'équipe « ne voulait pas utiliser les transformations opérationnelles », jugées « inutilement complexes » pour un outil de dessin, et a conçu un système que le billet présente comme inspiré des CRDT plutôt que comme un CRDT au sens strict : un serveur central ordonne les messages, chaque propriété d'un objet suit une règle « dernier écrivain gagne » où l'ordre du serveur remplace l'horloge, et les propriétés d'un objet supprimé ne sont pas conservées côté serveur. Lisez ce billet comme un témoignage d'ingénierie, pas comme un article de recherche : il montre surtout qu'une équipe emprunte les idées des CRDT à la carte, en gardant un serveur quand il simplifie la vie.

Enfin, Martin Kleppmann, Adam Wiggins, Peter van Hardenberg et Mark McGranaghan ont donné en 2019 un nom à l'ambition qui porte tout cela : le logiciel *local-first*. Leur manifeste énonce sept idéaux, dont la rapidité, le travail hors ligne, la collaboration et la propriété des données par l'utilisateur, et désigne les CRDT comme la brique technique qui les rend compatibles. Votre éditeur à deux répliques dans une VM en est une maquette honnête : la coupure est simulée, mais le raisonnement est le même.

8 Pour aller plus loin

Conflict-free Replicated Data Types, Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski (2011).

<https://inria.hal.science/inria-00609399>

L'article fondateur, court : la définition de la cohérence forte à terme, les deux familles de CRDT et la preuve que le demi-treillis suffit. Le rapport technique compagnon, « A comprehensive study », donne le catalogue des compteurs, ensembles et registres.

Conflict-Free Replicated Data Types (CRDTs), Nuno Preguiça, Carlos Baquero et Marc Shapiro (2018).

<https://arxiv.org/abs/1805.06358>

Une entrée d'encyclopédie de quelques pages par les mêmes auteurs, sept ans plus tard : la meilleure première lecture si l'article de 2011 vous paraît dense.

Data Consistency for P2P Collaborative Editing, Gérald Oster, Pascal Urso, Pascal Molli et Abdessamad Imine (2006).

<https://inria.hal.science/inria-00108523>

WOOT, le premier éditeur de texte « sans transformation ». À lire pour voir une autre manière d'ancrer une lettre, par ses deux voisins.

Replicated abstract data types: Building blocks for collaborative applications, Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim et Joonwon Lee (2011).

<https://doi.org/10.1016/j.jpdc.2010.12.006>

L'article de RGA, dont le projet est une adaptation. La section sur l'« array » répliqué décrit l'arbre d'ancres et l'ordre des frères que vous avez manipulés.

Interleaving anomalies in collaborative text editors, Martin Kleppmann, Victor Gomes, Dominic Mulligan et Alastair Beresford (2019).

<https://martin.kleppmann.com/papers/interleaving-papoc19.pdf>

Six pages, avec la figure 5 que cite la fiche du projet : l'arbre de RGA, son parcours, et le cas où il entrelace malgré tout.

The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing, Matthew Weidner et Martin Kleppmann (2023).

<https://arxiv.org/abs/2305.00583>

La suite : une définition formelle du non-entrelacement et un algorithme qui la satisfait. Plus exigeant ; lisez d'abord l'introduction et les figures.

Time, Clocks, and the Ordering of Events in a Distributed System, Leslie Lamport (1978).

<https://lamport.azurewebsites.net/pubs/time-clocks.pdf>

L'un des articles les plus cités de l'informatique, et l'un des plus lisibles. La relation « a pu influencer » et l'horloge en trois règles.

Local-first software: You own your data, in spite of the cloud, Martin Kleppmann, Adam Wiggins, Peter van Hardenberg et Mark McGranaghan (2019).

<https://www.inkandswitch.com/local-first/>

Le manifeste, écrit pour être lu d'une traite : pourquoi vouloir des applications qui marchent sans serveur, et ce que les CRDT y apportent.

9 Une question pour la route

Dans le projet, chaque geste est une insertion ou une suppression, et les trois lois tiennent parce que l'état ne fait que grandir. Supposez maintenant qu'Alice veuille

déplacer un mot, le couper ici et le coller là, pendant que Bob, hors connexion, corrige une lettre à l'intérieur de ce mot. Avec les seuls outils du projet, le déplacement devient une suite de suppressions et d'insertions nouvelles ; que devient alors la correction de Bob après fusion, et sur quelle ancre se retrouve-t-elle ? Quelle information faudrait-il ajouter à l'état pour qu'un déplacement soit un événement à part entière, et quelle loi de fusion faudrait-il alors vérifier à nouveau ? Il n'y a pas de corrigé : la question occupe encore des articles de recherche.

Pour toute question sur ce fascicule ou sur le projet, vous pouvez me contacter par email, comme pour les autres séances.